Utilización de DEVS para Evaluar Arquitecturas de Software

Verónica Bogado, Silvio Gonnet, Horacio Leone

INGAR, Universidad Tecnológica Nacional, CONICET
Avellaneda 3657, 3000 Santa Fe, Argentina
{vbogado, sgonnet, hleone}@santafe-conicet.gov.ar

Resumen. En el presente trabajo se propone un modelo para la simulación de productos de software en etapa temprana del desarrollo, empleando la arquitectura. El mismo se centra en la captura de la información necesaria relacionada al modelado arquitectónico y la transformación de los conceptos capturados a elementos de un modelo de simulación. Se propone el formalismo DEVS para incorporar las ventajas de la simulación en el contexto de diseño arquitectónico, ya que, a diferencia de otras herramientas de simulación, permite mantener el modelo desacoplado del simulador, y trabajar en forma modular y jerárquica. El modelo propuesto soporta la transformación de elementos arquitectónicos a elementos de un modelo de simulación, con el objetivo de obtener información cuantitativa para evaluar la calidad de un sistema en la etapa de diseño, permitiendo tomar decisiones tempranamente.

Palabras Claves: arquitectura de software, evaluación de arquitectura, DEVS

1 Introducción

Hoy en día, se sabe que los productos intermedios que se obtienen en las primeras etapas del desarrollo de software, como ser la especificación de los requerimientos y el diseño de la arquitectura, son claves para el producto final. El diseño de la arquitectura de software proporciona una base para analizar información que hace a la calidad del software. Sin embargo, realizar dicho análisis, más aun si se desea efectuar un "trade-off" entre atributos de calidad, requiere que el arquitecto adquiera conocimiento de diferentes técnicas dispares en uso y aplicación, resultando a las empresas dificultoso hallar recursos humanos con "know-how" necesario. Por tal motivo, resulta interesante el desarrollo de métodos y herramientas que den soporte al arquitecto de software en la etapa de diseño arquitectónico.

La arquitectura de software es la/s estructura/s del sistema, la cual comprende componentes de software, propiedades visibles externas de dichos componentes y la relación entre ellos; refleja atributos que el sistema debe contener [1]. La misma es una entidad compleja como para ser descripta en una única dimensión, por lo cual para realizar un buen estudio de la misma es necesario diferentes perspectivas o vistas, como las propuestas por Clements [2], Hofmeister [3] y Kruchten [4]. A pesar

de ser un tema no consensuado, los autores coinciden en modelar al menos una vista estática y una dinámica.

Por otro lado, es posible emplear Use Case Maps (UCM) para reflejar la parte funcional de un sistema en la arquitectura [5], mediante relaciones causales entre responsabilidades delimitadas dentro de elementos arquitectónicos, detallando así comportamiento y estructuras con un alto nivel de abstracción [6].

Las distintas representaciones de una arquitectura permiten evaluar atributos de calidad vinculados al software. Actualmente, se hallan trabajos que presentan propuestas para analizar la arquitectura con distintos propósitos. Algunos se orientan a la arquitectura (forma analítica centrada en los "stakeholders" y el experto), otros se enfocan en los atributos de calidad (análisis cualitativo o cuantitativo), o se focalizan en el cálculo de algún valor global que represente al atributo de calidad (medida de todo el sistema –Procesos de Decisión de Markov, Redes de Petri [7][8][9]). También existen trabajos basados en el uso de escenarios, dentro de los cuales se pueden encontrar aquellos orientados a patrones (análisis de la descripción arquitectónica realizado por un experto), los basados en decisiones de diseño (documentación y análisis de las decisiones tempranas que influyen en la evaluación de los atributos de calidad) y los basados en UCM (procesos de Markov y teoría de colas [10]).

Sin embargo, al momento de analizar la dinámica del sistema en etapas tempranas, se carece de herramientas que muestren cómo ocurren los cambios en el sistema. La simulación es un instrumento poderoso para analizar los estados por los que puede pasar un sistema y obtener valores para evaluar diferentes escenarios. Las variables que caracterizan una entidad, como ser valores que permitan medir la performance pueden ser modificados y estudiados, viendo el impacto que ello provoca sin necesidad de implementar el sistema.

Para ello, se plantea un modelo para la simulación de productos de software en etapa temprana del desarrollo de los mismos, empleando la arquitectura, de manera tal que permita el estudio del comportamiento, obteniendo medidas que sirvan para validar atributos de calidad especificados en los requerimientos del software.

El trabajo se centra en la captura de la información necesaria relacionada al modelado de la arquitectura de software y la transformación de los conceptos capturados a elementos de un modelo de simulación. Se propone emplear el formalismo Discrete Event System Specification (DEVS) [11], desarrollado por Bernard Zeigler a mediados de los 70, el cual ha sido probado ampliamente como herramienta para el modelado y simulación orientados a objeto. DEVS permite representar modelos en forma modular y jerárquica, provee un gran poder de expresión, abstracción y estructuración, lo cual se ajusta naturalmente a la forma en que los conceptos del dominio arquitectónico se relacionan.

2 Modelo Conceptual: Arquitectura de Software para Evaluar Atributos de Calidad

El modelo que se propone (ver Fig. 1) intenta capturar la información necesaria para hacer un análisis de la arquitectura desde un punto de vista dinámico, incluyendo datos para obtener resultados cuantitativos sobre diferentes aspectos. De esta manera,

poder comprender el comportamiento del sistema relacionado a los aspectos organizacionales de la arquitectura, secuencia de interacciones entre componentes que emergen en tiempo de ejecución. En el mismo se integran los elementos básicos necesarios para lo propuesto, partiendo de los diferentes enfoques existentes en la literatura ([1], [2], [3], [12]), incorporando información sobre la parte funcional del sistema mediante el uso de los UCM [6], siendo la base para la simulación posterior.

En una vista dinámica participan diferentes elementos arquitectónicos (*ElementoArquitectonico* en Fig. 1), los cuales son entidades de software que tienen presencia en tiempo de ejecución, diferenciándose: el componente (*Componente* en Fig. 1), entidad que tiene un conjunto de responsabilidades a cargo, y el mecanismo de conexión (*MecanismoConexion* en Fig. 1) que actúa entre 2 o más componentes, siendo la vía de interacción. Los componentes pueden ser simples (*ComponenteSimple* en Fig. 1) o compuestos (*ComponenteCompuesto* en Fig. 1). El simple no contiene otros componentes y conectores, es la unidad más pequeña dentro de los elementos arquitectónicos, y el compuesto está conformado de otros componentes, simples o compuestos, y sus conectores, siendo una unidad más compleja la cual delega sus responsabilidades en las partes.

Una responsabilidad (*Responsabilidad* en Fig. 1) es una declaración general sobre un objeto de software [13]: una acción que realiza el elemento, un conocimiento que mantiene sobre algo o una decisión importante que afecta a otro objeto de software. Un elemento arquitectónico puede tener asignado responsabilidades, las cuales se relacionan con otras del mismo elemento o de otros para realizar acciones requeridas. El tipo de relación es *causa-efecto* (*causa* en Fig. 1), donde el cumplimiento de una responsabilidad implica la ejecución de las consecutivas correspondientes [6].

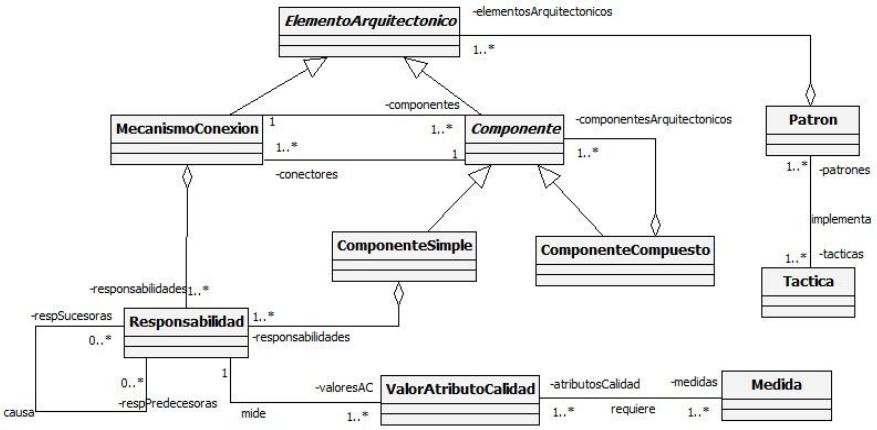

Fig. 1. Modelo Conceptual del Dominio Arquitectura de Software

Además de capturar los elementos que intervienen, resulta interesante que el arquitecto especifique valores relacionados a métricas relevantes para evaluar atributos de calidad, siendo importante documentar tanto las medidas (*Medida* en Fig. 1) como los valores de atributos de calidad (*ValorAtributoCalidad* en Fig. 1), los cuales pueden ser el resultado de una métrica simple o compuesta (varias medidas).

Finalmente, el arquitecto puede proponer mejoras, las cuales se implementan aplicando tácticas o patrones. Un patrón (*Patron* en Fig. 1) empaqueta un conjunto de tácticas (*Tactica* en Fig. 1), reuniendo elementos arquitectónicos para formar una unidad en sí con características que mejoren el modelo.

3 Jerarquía de Especificaciones DEVS para Elementos Arquitectónicos

A partir del modelo planteado (Fig. 1), se establecen las relaciones entre los elementos de dicho modelo y los elementos del modelo de simulación (ver Tabla 1).

El trabajo se centró en la construcción de modelos DEVS para las estructuras básica de un modelo arquitectónico (componente simple, mecanismo de conexión).

Elementos del Modelo Conceptual	Elementos del Modelo Simulación
Responsabilidad	DEVS Atómico con puertos
ComponenteSimple	DEVS Acoplado
MecanismoConexion	DEVS Acoplado
Relación: causa (entre responsabilidades)	Puertos E/S, acoplamiento entre puertos
Relaciones: agregación (ComponenteSimple-Responsabilidad, MecanismoConexion-Responsabilidad)	Acoplamiento entre puertos

Tabla 1. Relación entre modelos. Correspondencias entre elementos conceptuales (arquitectura de software) y elementos de simulación (jerarquía de modelos DEVS para el dominio)

Siguiendo los principios propuestos por Zeigler y otros autores [14], se construye un modelo de simulación, compuesto por modelos DEVS estructurados en forma modular y jerárquica, donde los elementos (bloques de construcción) se determinan en base a ciertas características, como ser que el elemento sea auto-contenido (información y procesos locales), interoperable (cooperación entre bloques), reusable (instanciación múltiple), reemplazable (intercambiable en el modelo), respetando el encapsulamiento de la estructura interna mediante interfaces bien definidas.

3.1 Modelo DEVS para Responsabilidad

El concepto *Responsabilidad* en el modelo conceptual (Fig. 1) se corresponde con MR en el modelo de simulación, un modelo DEVS atómico con puertos (Tabla 1).

Las relaciones entre responsabilidades se dan, en el modelo de simulación, a través de los puertos de entrada/salida. Cada modelo DEVS para *Responsabilidad* calcula sus valores de salida, tanto los datos de su estado como los necesarios para evaluar el atributo de calidad, teniendo en cuenta, por el momento, solamente el tiempo de ejecución, el cual puede servir para evaluar tiempos de respuestas del sistema ante ciertos estímulos y validar escenarios que se centren en la performance.

$$MR = \{X, Y, S, \delta_{ext}, \delta_{\text{int}}, \lambda, ta\}$$

3.1.1 Conjunto de Eventos de Entrada (X)

El conjunto de valores de entrada, representa información sobre las responsabilidades predecesoras e indica el grado de cumplimiento de cada responsabilidad predecesora. Por el momento, la única información necesaria para que se active es saber cuando finaliza la ejecución de la/s anterior/es. Aunque teniendo presente que dicho conjunto podrá crecer al incorporar mayor detalle en el modelo posteriormente.

Conjunto de puertos de entrada:

$PER = \{perp\}$ donde *perp* es el puerto de entrada al modelo MR que se conecta a puertos de salida de otros modelos MR o a puertos de entrada de modelos acoplados.

Conjunto de valores de entrada para el puerto *perp*:

$X_{perp} = \{finalizada\}$

Conjunto de puertos y valores de entrada:

$X = \{(perp, finalizada)\}$ donde $perp \in PER$

3.1.2 Conjunto de Estados (S)

Una responsabilidad refleja un punto donde el sistema realiza un cambio de estado debido a que es interrogado o afectado de alguna forma. Los estados permiten ver si el sistema se encuentra en este punto activo, en ejecución o en espera.

Fases:

- *Inactiva*: estado pasivo, en espera a que ocurra algún evento externo. Permanece indefinidamente así hasta que se produzca algo que interrumpa su estadío.
- *Activa*: estado transitorio, permite el disparo de una transición interna que genere una salida necesaria para la evaluación del sistema. Indica que la ejecución de la responsabilidad se ha iniciado. Su duración es nula, tal que los eventos externos no puedan intervenir.
- *Ejecutando*: indica que la responsabilidad se está llevando a cabo, donde la ejecución se refiere al procesamiento de un bloque de código (software).

σ: tiempo restante en el estado dado.

$$S = \{\text{inactiva, activa, ejecutando}\} \times R_0^+$$

Parámetro (Fijo del modelo): *tiempo_ejecucion,* tiempo que demora en llevar a cabo la responsabilidad un elemento arquitectónico, según alguna distribución.

3.1.3 Conjunto de Eventos de Salida (Y)

Una responsabilidad debe ser encargada de emitir dos tipos de información, uno relativo a su estado, información de interés para las responsabilidades sucesoras, y otro referido a los valores usados para medir aspectos sobre atributos de calidad.

Conjunto de puertos de salida:

$PSR = \{psrs, psm\}$ donde

psrs: Puerto de salida de eventos hacia responsabilidades sucesoras

psm: Puerto de salida de medidas para evaluar el atributo de calidad (performance)

Conjunto de valores para el puerto *psrs*:

- *Activada*: se inicia la realización de la responsabilidad, lista para la ejecución.
- *Finalizada*: se ha llevado a cabo la responsabilidad, la ejecución terminó.

$Y_{psrs} = \{activada, finalizada\}$

Conjunto de valores para el puerto *psm*:

$Y_{psm} = R_0^+$ indica el tiempo que demoró la ejecución de la responsabilidad

Conjunto de puertos y valores de salida:

$Y = \{(psrs, activada), (psrs, finalizada), (psm, m)\}$ donde $m \in Y_{psm}$, $psrs \in PSR$, $psm \in PSR$

3.1.4 Función de Transición Interna

Determina el siguiente estado de la responsabilidad como resultado del paso del tiempo, no de un evento externo. El estado transitorio "*activa*" indica que la responsabilidad puede ser llevada a cabo, ya que sus predecesoras han sido cumplidas correctamente. Informa a los elementos del modelo de simulación correspondientes que la misma se activó, para ello es necesaria la transición de estado interna, permitiendo emitir un evento por el puerto correspondiente, y pasando automáticamente al siguiente estado "*ejecutando*". La otra transición interna se produce luego de acabado el tiempo de ejecución que requiere la responsabilidad para ser cumplida, regresando al estado "*pasivo*", en espera de otro evento externo.

$\delta_{int}(ejecu\ tan\ do, \sigma) = (inactiva, \infty)$

$\delta_{int}(activa, \sigma) = (ejecu\ tan\ do, tiempo\_ejecucion)$

3.1.5 Función de Transición Externa

Esta función efectúa una transición de estados cuando se produce un evento externo, cuando se recibe el valor "*finalizada*" de todas las responsabilidades predecesoras.

$$\delta_{ext}(fase, \sigma, e, x) \begin{cases} (activa, 0) & si\ fase = pasiva \\ (fase, \sigma - e) & si\ fase = otra \end{cases}$$

3.1.6 Función de Salida

La función de salida emite un valor, luego se realiza la transición interna de estados.

$\lambda(activa, \sigma) = (psrs, activada)$

$\lambda(ejecu\ tan\ do, \sigma) = (psrs, finalizada) \wedge (psm, m)$ donde $m \in Y_{psm}$

3.1.7 Función de Avance del Tiempo

Determina el tiempo que se permanece en el estado. Para el modelo: $ta(s) = \sigma$

$ta(activa, \sigma) = 0$ siendo s un estado transitorio

$ta(ejecu\ tan\ do, \sigma) = tiempo\_ejecucion$ (especificado para el modelo)

$ta(inactiva, \sigma) = \infty$ siendo *s* un estado pasivo

3.2 Modelo DEVS para Componente Arquitectónico Simple

El componente simple tiene a cargo un conjunto de responsabilidades. La relación que existe entre el mismo y sus responsabilidades se puede modelar en DEVS como una jerarquía donde los modelos DEVS de las responsabilidades (MR) son componentes del modelo DEVS acoplado que representa el componente simple arquitectónico.

$$CS = (X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC)$$

3.2.1 Conjunto de Puertos y Valores de Entrada (X)

El modelo acoplado CS tiene un conjunto de puertos de entrada los cuales propagan los valores a sus componentes (elementos del modelo de simulación).

Conjunto de puertos de entrada:

$PECS = \{peep\}$ donde *peep* es el puerto de entrada que recibe información de elementos predecesores y se conecta a puertos de entrada de modelos componentes.

Conjunto de valores de entrada:

El conjunto de valores de entrada, representa información sobre elementos arquitectónicos predecesores. La información necesaria para activar la ejecución del componente es la referida a la finalización de la ejecución de elementos anteriores.

$X_{peep} = \{finalizada\}$

Conjunto de puertos y valores de entrada:

$X = \{(peep, finalizada)\}$ donde $peep \in PECS$

3.2.2 Conjunto de Puertos y Valores de Salida (Y)

Propaga los eventos de los modelos componentes del mismo a otros componentes.

Conjunto de puertos de salida:

$PSCS = \{pses, psm\}$ donde

pses : Puerto de salida de eventos hacia elementos sucesores

psm : Puerto de salida de medidas para un atributo de calidad (performance)

Conjunto de valores posibles de salida:

Los valores de salida para el primer puerto (*pses*):

- *Activada*: se inicia la ejecución del componente.
- *Finalizada*: se ha terminado la ejecución del componente.

$Y_{pses} = \{activada, finalizada\}$

Para el puerto *psm*, el valor de salida es un número que indica alguna medida.

$Y_{psm} = R_0^+$ indica el tiempo que demoró la ejecución del componente

Conjunto de puertos y valores de salida:

$Y = \{(pses, activada), (pses, finalizada), (psm, m)\}$ donde $m \in Y_{psm}$, $pses \in PSCS$, $psm \in PSCS$

3.2.3 Conjunto de componentes (D)

Detalla el conjunto de referencias de los componentes (MR) del modelo acoplado.

3.2.4 Componentes (modelos)

$CR_j = \text{MR} \quad \forall j \in D$ donde *j* es el nombre de referencia del modelo.

3.2.5 Acoplamientos

Acoplamiento Externo de Entrada (EIC):

$EIC \subseteq \{((CS, peep_{CS}), (r, perp_r)) \mid r \in D, peep_{CS} \in PECS, perp_r \in PER_r\}$

Acoplamiento Externo de Salida (EOC):

$EOC \subseteq \{((r, psrs_r), (CS, pses_{CS})) \mid r \in D, pses_{CS} \in PSCS, psrs_r \in PSR_r\}$

Acoplamiento Interno (IC):

$IC \subseteq \{((r_1, psrs_{r_1}), (r_2, perp_{r_2})) \mid r_1, r_2 \in D, psrs_{r_1} \in PSR_{r_1}, perp_{r_2} \in PER_{r_2}\}$ donde $r_1 \Rightarrow r_2$

$((r_1, psrs_{r_1}), (r_2, perp_{r_2})) \in IC \Rightarrow r_1 \neq r_2, \quad \forall r_1, r_2 \in D$

Observación: el modelo DEVS para el Mecanismo de Conexión se especifica similarmente al CS, con sus respectivos puertos de E/S, acoplando modelos MR.

3.3 Implementación en DEVSJAVA

DEVSJAVA [15] es un conjunto de bibliotecas para implementar los modelos DEVS usando el lenguaje de programación JAVA. El mismo brinda el paquete *Zdevs*, que contiene la clase *devs*, base (superclase) de los dos modelos principales (atómico-clase *atomic* - y acoplado - clase *coupled,* de la cual se desprende la clase *digraph*). Las implementaciones de las clases *ResponsibilityM (atomic)*, *SimpleComponentM (digraph)*, *ConectionMechanismM* (*digraph*) corresponden a los modelos propuestos, siendo clases heredadas de las que provee DEVSJAVA.

4 Ejemplo de Aplicación

A modo de ejemplo se expone un caso clásico de la arquitectura de software, patrón MVC (Model-View-Controller), donde se muestra una vista de alto nivel usando un UCM (ver Fig. 2). La definición de las responsabilidades en el mapa depende del grado de detalle que se desea reflejar, intentando evitar introducir particularidades que obscurezcan el entendimiento, dado el nivel de abstracción con que se trabaja en el diseño arquitectónico [16].

En el caso de estudio, el controlador es el encargado de gestionar los eventos que se producen como consecuencia de los requerimientos realizados por el usuario (responsabilidad 1 en Fig. 2) y, en consecuencia, comunicarse con el modelo para actualizarlo de acuerdo a la acción solicitada (responsabilidad 2 en Fig. 2). El modelo es la representación de la información con la cual el sistema opera y es el encargado

de cambiar dicha información en sí (responsabilidad 3 en Fig. 2), recibiendo las órdenes de modificación desde el controlador. Luego, se delega a la vista la tarea de desplegar la interfaz del usuario (responsabilidad 4 en Fig. 2), siendo quien obtiene sus datos del modelo para generar dicha interfaz donde se reflejen los cambios realizados en el modelo.

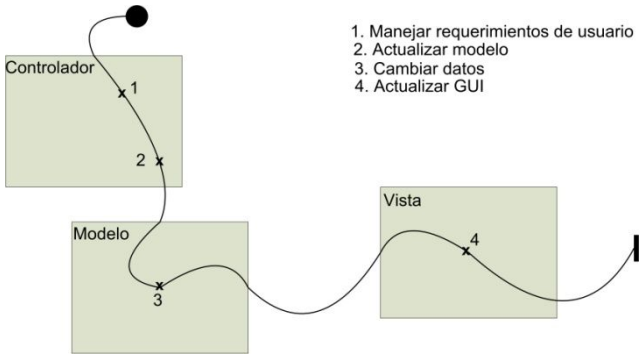

Fig. 2. UCM para el patrón MVC (adaptado de [16])

Los tres componentes arquitectónicos se traducen a modelos DEVS acoplados (CS) y las responsabilidades a modelos DEVS atómicos (MR), describiendo cómo se vinculan los puertos y se estructuran los componentes DEVS para el componente *Controlador* (CS1) en la Fig. 3. Se muestra cómo se acoplarían las responsabilidades de dicho componente arquitectónico en el modelo de simulación, las conexiones de sus puertos con los correspondientes. Cada responsabilidad, al ser un modelo atómico *MR*, estaría a cargo de emitir las salidas correspondientes y calcular la información necesaria para evaluar aspectos de la arquitectura (tiempo de ejecución considerado actualmente en el modelo). El modelo acoplado (CS1) propagaría la información hacia sus componentes (mr1, mr2) y hacia el exterior, según sean entradas o salidas.

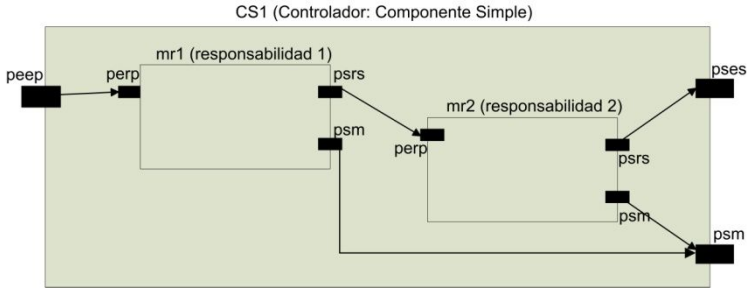

Fig. 3. Componente Controlador (CS). Puertos y componentes internos (MR).

5 Conclusiones

En el presente trabajo se propuso un modelo conceptual que permite capturar información tanto de los elementos básicos que intervienen en una arquitectura de

software como información de su comportamiento (responsabilidades), y valores cuantitativos que permiten analizar atributos de calidad, de manera tal que se pueda disponer de información necesaria como para validar diferentes escenarios.

Para complementar el modelo se presentó un formalismo (DEVS) que incorpora las ventajas de la simulación en el contexto de diseño arquitectónico. Dicho formalismo, a diferencia de otras herramientas de simulación permite mantener el modelo desacoplado del simulador, y trabajar los modelos en forma modular y jerárquica. Se describió la transformación de los conceptos del modelo (información de la arquitectura) a elementos del modelo de simulación del formalismo propuesto.

Como trabajos futuros se pretende especificar e implementar los demás conceptos del dominio (componentes compuestos y patrones), agregándolos al modelo de simulación. Además, resulta interesante seguir profundizando sobre los aspectos cuantitativos del modelo arquitectónico, incorporando parámetros a los bloques del modelo de simulación para poder evaluar atributos visibles en tiempo de ejecución.

Referencias

1. Bass, L.; Clements, P.; Kazman: Software Architecture in Practice. Addison-Wesley, (2003).
2. Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Nord, R.; Stafford, J.: Documenting Software Architecture- Views and Beyond. Addison-Wesley, (2002).
3. Hofmeister, C.; Nord, R.; Soni, D.: Applied Software Architecture. Addison-Wesley, (2000).
4. Kruchten. P,: "The 4+1 View Model of Architecture," IEEE Software, vol. 12, (1995).
5. Buhr, R.: “Making Behaviour a Concrete Architectural Concept”. Proceeding of 32nd Hawaii International Conference on System Sciences, (1999).
6. Buhr, R.; Casselman, R.: Use Case Maps for Object-Oriented Systems. Prentice Hall (1999).
7. Wang, W.; Wu, Y.; Chen, M.: “An Architecture-Based Software Reliability Model”. In: Pacific Rim International Symposium on Dependable Computing, 143-150, (1999).
8. Fukuzawa, K..; Saeki, M.: “Evaluating Software Architecture by Coloured Petri Nets”. Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, (2002).
9. Spitznagel, B.; Garlan, D.: “Architecture-Based Performance Analysis”. Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering , (1998).
10. Petriu, D.; Woodside, M.: “Software Performance Models from System Scenarios in Use Case Maps”. Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools, (2002).
11. Zeigler, B., Praehofer, H., Kim,. T.: Theory of Modeling and Simulation–Integrating Discrete Event and Continuous Complex Dynamic Systems. Academic Press, (2000).
12. ISO/IEC WD3 42010. IEEE P42010/D3. Systems and Software Engineering – Architectural Description, (2008).
13. Wirfs-Brock, R.; McKean, A.: Object Design: Roles, Responsibilities and Collaborations. Addison-Wesley, (2002).
14. Verbraeck, A.; Valentin, E.: “Design Guidelines for Simulation Building Blocks”. Proceedings of the 2008 Winter Simulation Conference (WSC), (2008).
15. Zeigler, B., Sarjoughian, H.: “Introduction to DEVS Modeling an Simulation with JAVA: Developing Component-Based Simulation Models”, (2005).
16. Buhr, R.; Casselman, R.; Pearce, T.: “Design Patterns with Use Case Maps: A Case Study in Reengineering an Object-Oriented Framework”, reporte técnico, SCE 95-17, (1996).